

Parallel Discrete Event Simulation Course #7

David Jefferson
Lawrence Livermore National Laboratory
2014

LLNL-PRES-652107

This work was performed under the auspices of the U.S. Department
of Energy by Lawrence Livermore National Laboratory under Contract
DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Release Number: LLNL-PRES-652107

Reprise

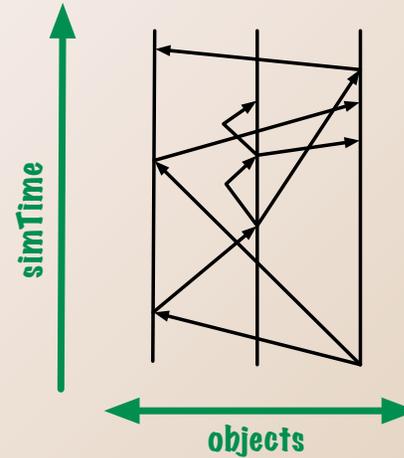
Critical Path Theory

How fast can a particular simulation run on a particular machine?

How much parallelism is present in the application?

Causality digraph and parallelism

- Every event (except initial) is scheduled by some other event
- Vertical arcs: successive state changes in one object
- Diagonal arcs: event messages
- Paths represent *causality chains*
- Events connected by a path must (appear to) be executed **sequentially**
- Events not connected by a path can be executed **in parallel**, even if out of time order.

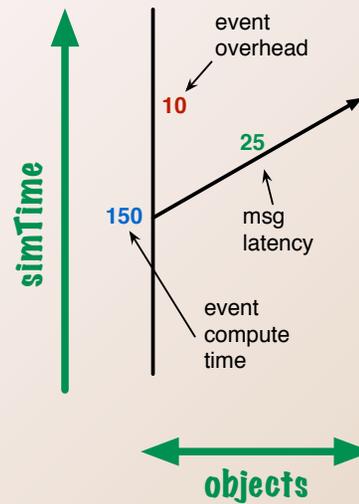


Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

4

A PDES can be viewed as a digraph in simulation spacetime. In this diagram the horizontal axis is the space of objects, and the vertical axis is simulation time. The nodes in the graph are events, and the arcs are *causal connections* representing information flows. The vertical arcs (without arrow heads) represent the flow of an object's state from one event to the next in that object. The slanted arcs represent event messages sent by one event to be received during another.

Add wall clock timings to spacetime digraph



Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

5

The numbers here represent wall clock times on a particular (real or idealized) platform.

The **blue** numbers on the events represent the wall clock time taken to execute an event. I have included a range of event execution lengths in this diagram, anywhere from 45 to 200 units of time.

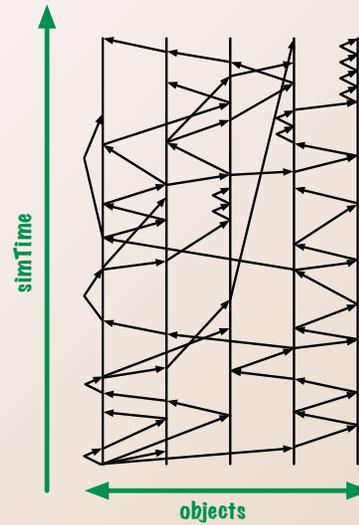
The **red** numbers on the arcs between events in the same LP represent event overhead, i.e. all of the computation that is not direct execution of events. It includes synchronization, storage management, etc. In this case I have made the arbitrary assumption that that is a constant 10 units of time everywhere.

The **green** numbers on the arcs that represent event messages indicate the message latencies, i.e. the wall clock time required to transport a message from one process to another. In this case I have made the arbitrary assumption that it takes 25 time units to send a message between two neighboring LPs, and more time if the communicating LPs are more distant.

In each case these numbers should either be estimates of the minimum times each of these should take, or the average times. Use minimum times if you wish to get an actual lower bound on runtime of the entire simulation. Use average times (considering message traffic effects and LP scheduling effects) if you want a somewhat more accurate estimate that is not strictly a lower bound.

If the event overheads are negligible then you can set them to zero. If the message latency is negligible (e.g. in shared memory) then you can set them to zero. Etc.

Space-Time Diagram of PDES

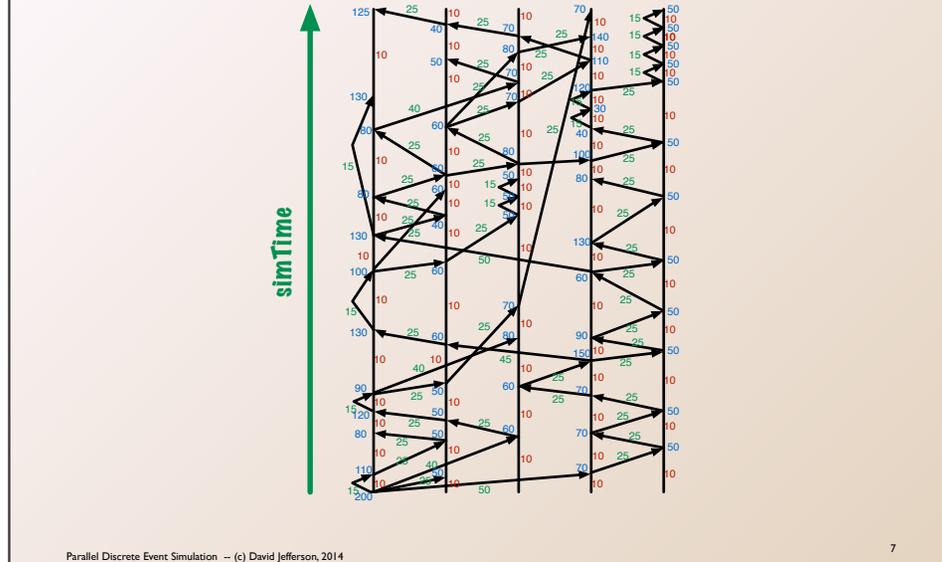


Parallel Discrete Event Simulation – (c) David Jefferson, 2014

6

This space-time graph depends only on the code and the input initialization of the simulation. The events executed, the simulation times at which they occur, and the causal relations among them are all deterministic. So this graph is the same regardless of timing, synchronization, platform or runtime configuration.

Space-time Digraph Labeled with Timings



The numbers here represent wall clock times on a particular (real or idealized) platform.

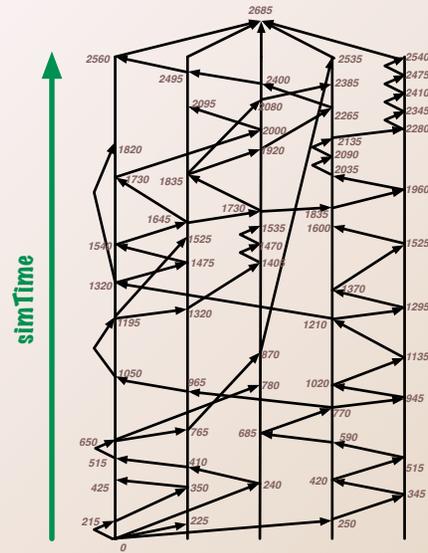
- The **blue** numbers on the events represent the wall clock time taken to execute an event. I have included a range of event execution lengths in this diagram, anywhere from 45 to 200 units of time.
- The **red** numbers on the arcs between events in the same LP represent event overhead, i.e. all of the computation that is not direct execution of events. It includes synchronization, storage management, etc. In this case I have made the arbitrary assumption that that is a constant 10 units of time everywhere.
- The **green** numbers on the arcs that represent event messages indicate the message latencies, i.e. the wall clock time required to transport a message from one process to another. In this case I have made the arbitrary assumption that it takes 25 time units to send a message between two neighboring LPs, and more time if the communicating LPs are more distant.

In each case these numbers should either be estimates of the minimum times each of these should take, or the average times. Use minimum times if you wish to get an actual lower bound on runtime of the entire simulation. Use average times (considering message traffic effects and LP scheduling effects) if you want a somewhat more accurate estimate that is not strictly a lower bound.

When you consider execution on a particular platform with a particular (static) assignment of objects to nodes, then the actual event timings, event overheads, and message latencies are introduced, based on the best case performance on that platform and in that configuration.

Note that this applies to STATIC configurations. If there is dynamic load reconfiguration of any kind, then this diagram does not capture such effects.

Critical Times



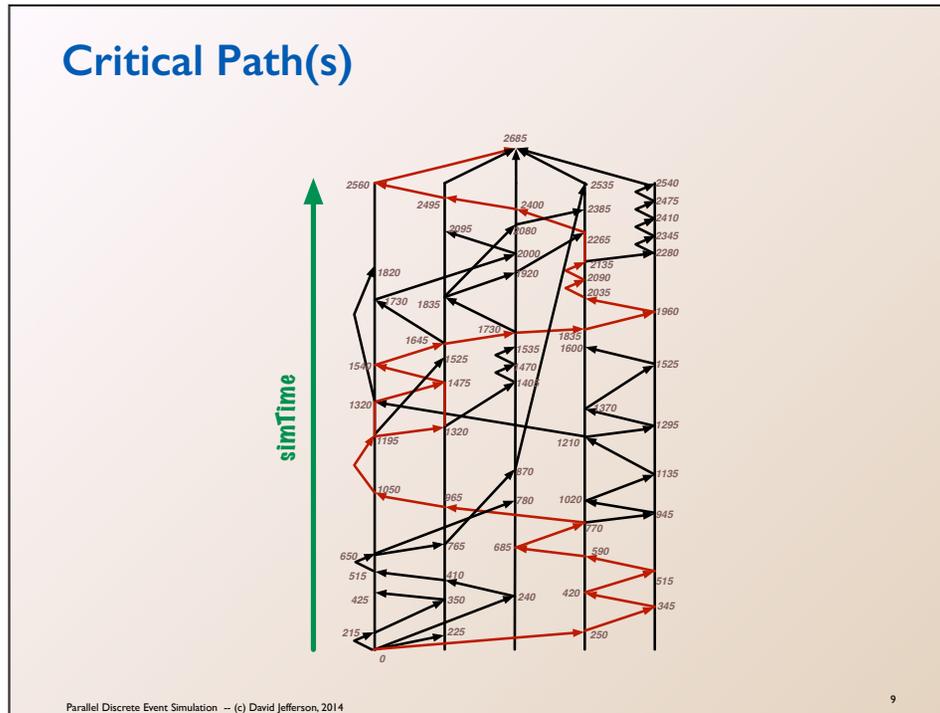
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

8

These critical times are for the start of the events that they are attached to. Thus, if an event has critical time 650, then it cannot possibly start execution until wall clock time 650. For simplicity we assume that when an event sends one or more messages, it does so at the end of the event's execution, even though in reality it might happen in the middle of the execution. Note that in the diagram we have added a final fictitious termination event.

The critical time for each event can be calculated on the fly during a run of the simulation. It takes negligible, constant time overhead per event to do so.

Critical Path(s)



Parallel Discrete Event Simulation – (c) David Jefferson, 2014

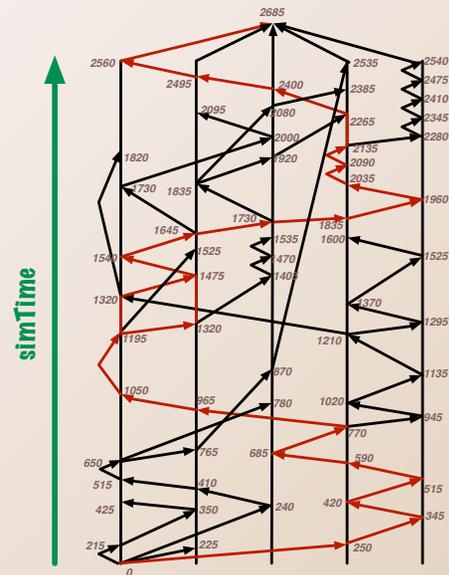
9

The critical path is not necessarily unique. There may be several, or many, paths that tie as the longest path. This diagram shows two paths of equal length, which are identical except for two alternative segments in the middle of the timelines of left two objected. More generally, there may be many paths of near-equal length, so that they are all co-critical.

The critical times and critical path can be calculated on the fly, during the simulation, in time linear in the length of the simulation, i.e. a constant time overhead per event. So it can and should be done routinely during PDES simulations. The actual critical path(s) though, requires taking a trace of the computation and post-processing it, and so is not usually done.

Co-critical Path(s)

- Many critical and near-critical paths -- **co-critical paths** -- in a large simulation.
- Performance improvement requires shortening all co-critical paths -- without lengthening others to criticality.
- Critical times and co-critical paths can be computed during a run with linear overhead
- In improving performance, consider:
 - code shared among many co-critical paths -- improve that shared code
 - objects shared among co-critical paths -- improve those objects
 - critical latencies between pairs of objects -- shorten the distances between those objects



Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

10

There does not have to be a unique critical path in a simulation. There can be multiple paths that tie for being the longest. But more importantly, there can be many, many paths that are very close to the same length as the one that is absolutely the longest. In order to improve the performance of a discrete event simulation one has to modify the code or the configurations so that *all of the critical and near critical paths are shortened*.

Speed-up from parallelism

n	number of objects (LPs) in the simulation
p	number of processors used
T _{seq}	sequential execution time
T _{crit}	critical path length
T _p	actual measured time on p nodes of cluster
S _{actual(p)}	actual measured speedup from parallelism using p nodes
S _{potential}	potential speedup from parallelism (generalized Amdahl's law)
S _{fraction(p)}	fraction of maximal speedup that is actually achieved
E _{procs(p)}	processor efficiency; fraction of processor time used for (committed) event execution

n	=	5		
p	=	5		
T _{seq}	=	4725		
T _{crit}	=	2685		
T _p	=	3200 (suppose)		
S _{actual(p)}	=	T _{seq} / T _p	=	4725/3200 = 1.48
S _{potential}	=	T _{seq} / T _{crit}	=	4725/2685 = 1.76
S _{fraction(p)}	=	S _{actual(p)} / S _{potential}	=	1.48/1.76 = 0.84
E _{procs(p)}	=	T _{seq} / (T _p * p)	=	4725/(3200*5) = 0.295

Parallel Discrete Event Simulation – (c) David Jefferson, 2014

11

Critical path analysis of parallel algorithms is really a generalization of the familiar Amdahl's law regarding the amount of potential speedup that is possible in an application when some of it has to be executed sequentially. The critical path in a distributed computation is the longest causal path that has to be executed sequentially, and this it determines the limit on the speedup that is possible from parallelism.

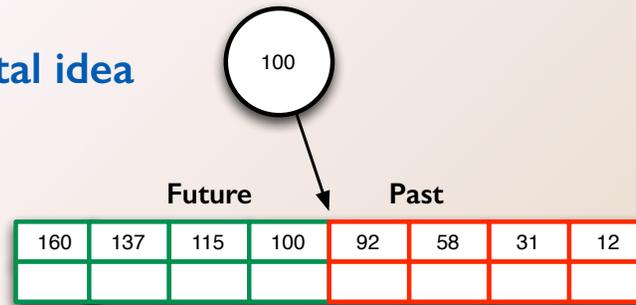
(Actually, the critical path limit bounds the speedup that can be achieved with any conservative method and with some optimistic methods. But surprisingly, there are optimistic methods that can actually beat the critical path lower bound! Time Warp with lazy cancellation was the first one known, but there are several other variants. This will be discussed in a future lecture.)

Optimistic Parallel Discrete Event Simulation Algorithms

Optimistic Paradigm

- **Events are considered reversible -- they can be undone (rolled back).**
- **Rollback is the fundamental synchronization primitive, not process blocking.**
- **No static restrictions on model structure are needed**
 - Any object can send event messages to any other at any future time
 - No graph structure or “channels” required or useful — just a flat name space
 - Order preservation during message transport not required
 - Dynamic object creation and destruction permitted
- **No lookahead information needed**
- **Optimistic PDES first introduced with the Time Warp algorithm in 1984 (RAND Corp., JPL)**
- **Many variants now, all descended from TW**
- **Many beautiful symmetries and analogies to physics**

Fundamental idea



- Assume infinite storage (for now)
- Save all event messages in one input queue, both processed and unprocessed
- Object `simTime` acts as a pointer into message queue, dividing past from future
- Process all events in `simTime` order
- Block only when all events in the queue have been processed, in which case set `sim clock` to ∞

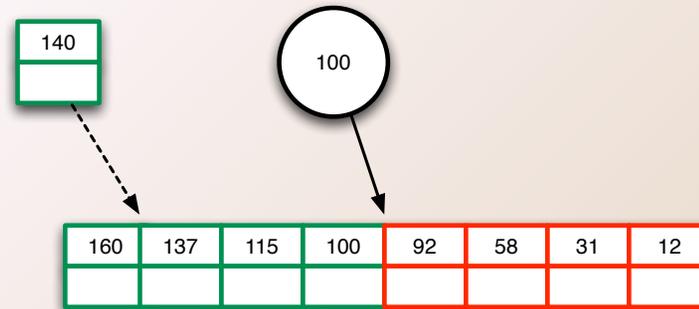
Parallel Discrete Event Simulation – (c) David Jefferson, 2014

14

An object at simulation time 100, along with its input queue. The input queue of simulation clock acts as a “pointer” into the message queue, dividing messages already processed (the “past”) from those not yet processed (the “future”). Messages in the past are not deleted even though they have been processed because we may have to roll back and reprocess them.

For the time being we will assume infinite storage, so there is no limit on the number of past messages we can handle. We will fix this limitation later.

Fundamental idea

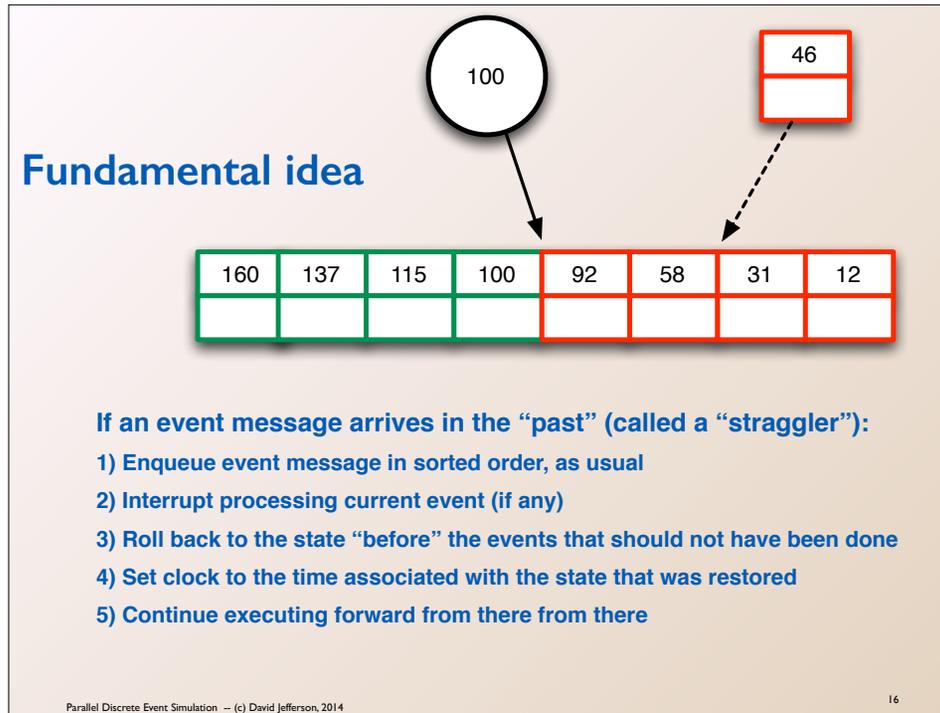


If an event message arrives in the “future”, just enqueue it in sorted order and continue processing events

Parallel Discrete Event Simulation – (c) David Jefferson, 2014

15

If an event message arrives with a time stamp in the future (i.e. greater than the object's current simulation clock) then just enqueue it in its proper sorted position and continue processing events.



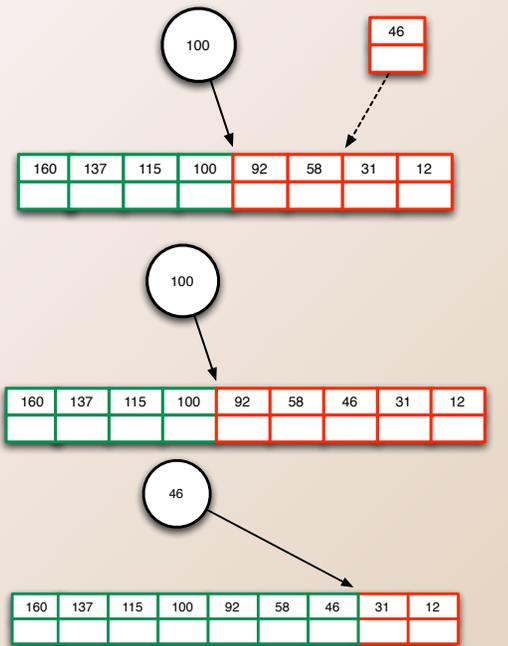
If an event comes in with a time stamp in the past however, we call it a “straggler”, and this calls for rollback.

- 1) Enqueue the message in its proper place in the queue, as usual. (Note, as a forward reference, this rule will hold regardless of whether we are talking about an input queue, an output queue, or a state queue, and regardless of whether the message is positive or negative.
- 2) It is best if we do not wait for the currently executing event to complete before the rollback commences (although most implementations do), but instead interrupt it. This is because the event is probably executing “incorrectly”, based on a state it should not be in because it did not execute the straggler message (timestamp=46 in this case), and the programmer never anticipated executing from this state, so as a result, the event code may be in an infinite loop, or if not, may still be executing a for horrendously long time, whereas a “correct” event execution in a “correct” state was designed to terminate quickly.
- 3) To complete the rollback, we have to restore the state that we were in when the straggler message should have been processed, in this case the state after processing of message 31. All side effects of processing message 58 and 92, and those of the partially processed message 100, must be reversed, including messages event messages sent, I/O operations, the freeing of storage, etc.
- 4) Set simulation clock of this object to that in the time stamp of the incoming message.

Note that if a message arrives in the “present”, i.e. with a time stamp exactly equal (to the last bit) to the one we are currently executing, in this case 100, then this is a tie. We have to roll back to before we started The current event, and then invoke our tie-breaking rule, whatever it is, in order to proceed forward.

Synchronization using

If an event message arrives in the “past”, roll back to “before” the event the events that should not have been done, and restart from there



Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

17

This diagram show that with the arrival of a straggler with time stamp 46, we must rollback the object to (at least) time 46, and start executing forward from there. Messages that used to be in “past” and already processed are suddenly in the “future” and considered unprocessed.

Asynchronous, distributed rollback? Are you serious???

- **Must be able to restore any previous state (between events)**
- **Must be able to cancel the effects of all “incorrect” event messages that should not have been sent**
 - even though they may be in flight
 - or may have been processed and caused other incorrect messages to be sent
 - to any depth
 - including cycles!
- **Must do it all *asynchronously*, with *many concurrently interacting rollbacks in progress*, and *without barriers***
- **Must deal with the consequences of executing events starting in “incorrect” states**
 - runtime errors
 - infinite loops
- **Must deal with truly irreversible operations**
 - I/O, or freeing storage, or launching a missile
- **Must be able to operate in finite storage**
- **Must guarantee global progress (if sequential model progresses)**
- **Must achieve good parallelism, and scalability**

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

18

This shows the list of challenges we have to overcome for the Time Warp algorithm, or any optimistic PDES algorithm, to be practical. Most people with a background in asynchronous distributed computation who have not seen optimistic PDES algorithms are inclined to believe that doing this is either literally impossible, or at least hopelessly complex and slow.

The Time Warp Algorithm

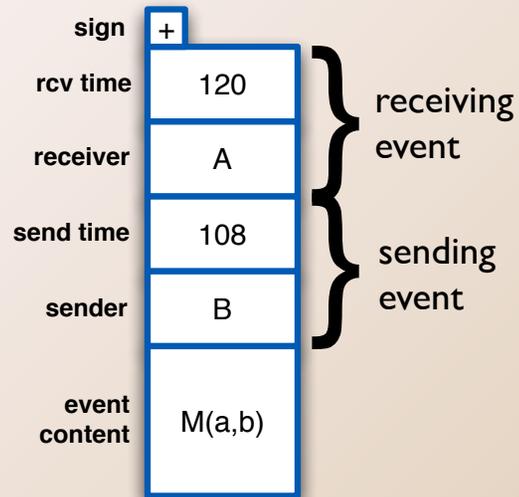
Brief History of Time Warp

- Invented by myself and Henry Sowizral at the RAND Corp. in 1983
- Implemented and studied at JPL on Caltech and JPL Hypercubes from 1985-1991
- Many other contributors in the early years (Brian Beckman, Peter Reiher, Anat Gafni, Orna Berry, Richard Fujimoto, just to mention my immediate associates, along with many others nationally)
- First journal publication:

Jefferson, David, "Virtual Time", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 7, 3, pp. 404-425, July 1985

Time Warp Algorithm

- (sender, sendtime) are spacetime coordinates of sending event
- (receiver, rcvtime) are spacetime coordinates of receiving event
- sign is + or -
- messages identical, but with opposite sign are “antimessages”



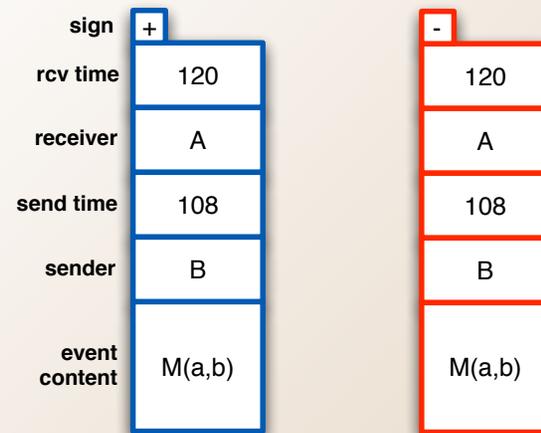
Parallel Discrete Event Simulation – (c) David Jefferson, 2014

21

An event message in the TW algorithms holds the spacetime coordinates of the sending event (B,108) and the spacetime coordinates of the receiving event (A,120). It also has a sign, + or -, whose purpose will become apparent.

For reasons of symmetry, we consider saved states (forward reference) to also have a “send time” and a “receive time”. The “send time” is the time of the event that produced the state, and the “receive time” is the time of the event that consumes the state, i.e. the time of the next event.

Message and Antimessage

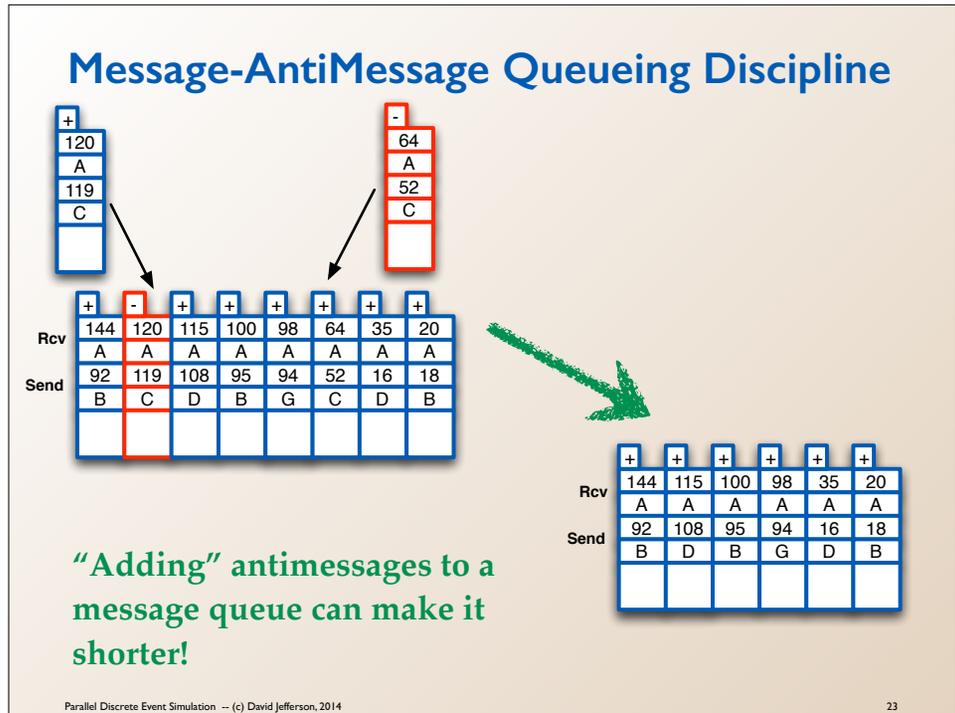


Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

22

Two event messages that are identical in all respects except for their signs are “antimessages” of one another.

As we will see, this terminology is apt because if antimessages come into “contact” with one another (by being enqueued in the same queue) they mutually annihilate. More generally, as we will see later, messages are always created in message-antimessage pairs and annihilated in message-antimessage pairs.



This is a “before” and “after” diagram. The message queue before (which happens to be an input queue since it is sorted by receive time) has 8 messages in it. (One of those messages is negative--that can happen, although it is infrequent).

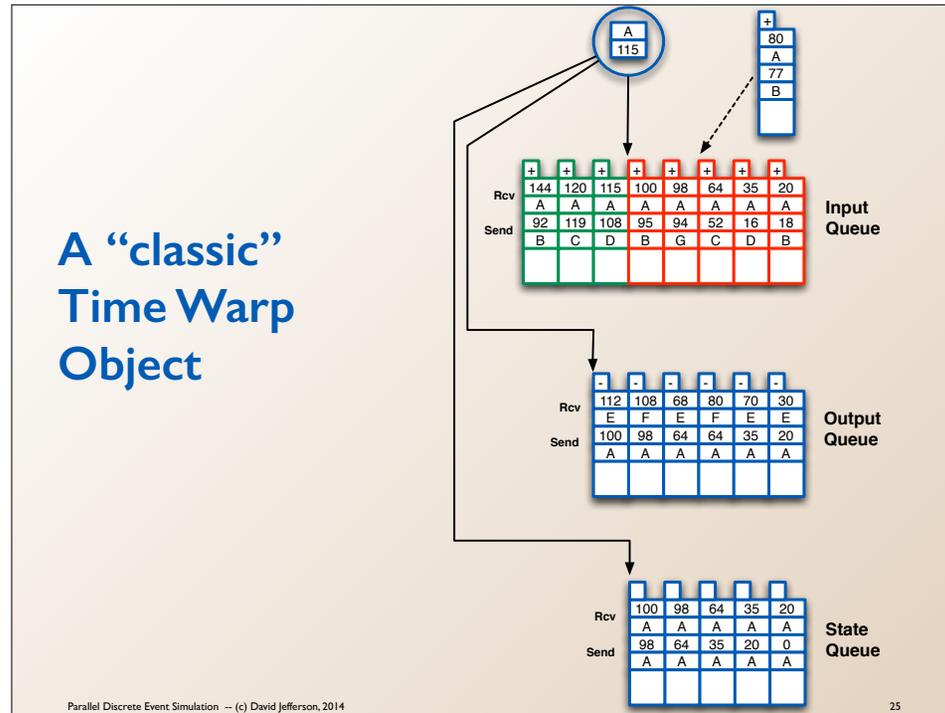
Two new messages arrive for enqueueing that happen to be antimessages to two other already in the queue. When they are “added” to the queue, the result is two annihilations and the queue gets shorter.

As far as I know this is the only “collection” data type appearing in CS literature which admits of “negative” objects like this, so that “adding” to a collection results in the collection getting smaller. It is not the same as just adding a delete(element) method to the data type, because such a method has no effect if the element is not present in the collection, whereas adding an antielement does have an effect, which is manifest either immediately to annihilate with its counterpart, or later if and when its counterpart is enqueued

Note also, that it is not forbidden to have two or more identical copies of the same event message in a queue--that constitutes a tie and calls for invocation of a tie breaking rule, but is otherwise OK. If, later an antimessage of one of them arrives, it annihilates with only one of the messages, leaving the others present. Message-antimessage annihilation “conserves charge”.

The Time Warp algorithm with its message-antimessage terminology could, of course, be re-described without the colorful elementary particle analogy. However, I think it helps to reveal the symmetries of the algorithm--there will be many more to come. So if you begin to think that the analogy is a little strained, please hold off until you see the way it develops later, particularly when it comes to the flow control and storage management parts of the TW algorithm.

End Reprise



Event 115 is executing at the moment

Each incoming and outgoing event message, and each state, is labeled with the spacetime coordinates of its “sending” and “receiving” events.

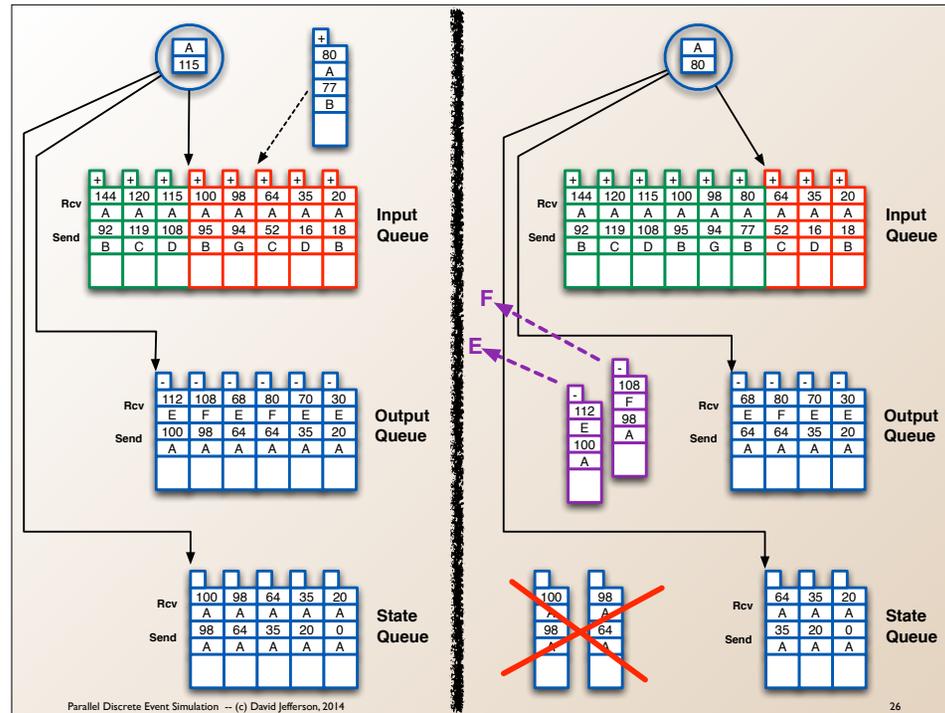
Input queue contains incoming event messages and is sorted by Receive Time. (Send Times are not necessarily in sorted order.) Input messages are usually all positive (but not always). The input queue contains both past (processed) and future (unprocessed) messages

Output queue contains outgoing event messages (negative copies) and is sorted by Send Time. Output messages are generally all negative (but not quite always—more later about that). The output queue often contains only past messages (but in later variations we will correct that to allow “future” output messages in certain interesting TW variations, e.g. “lazy cancellation”).

State Queue contains snapshots of states take between events, and is sorted by both send time and receive time-- the sorting is identical either way, so there is no distinction. The snapshots are only of static memory, and sometimes heap memory -- not stack memory, as the stack is always empty between events

Saved states are all “neutral”. Actually, this is something of a stretched convention, but it makes some sense, since a state is both an output from one event and an input to the next. The state queue contains only past states (but in later variations we will consider variations of TW in which “future” states make sense).

The reason I mention that in some variations of TW there are such things as “future output messages” (sent in the future) and “future states” (generated in the future), is to emphasize the symmetry of the TW object representation and algorithm among the input queue, output queue, and state queue of a TW object. The future states or future output messages are less frequently useful than the obvious desirability of future input messages—but this is just a performance issue. Logically, all three queue are symmetric, but the symmetry is broken by performance considerations.



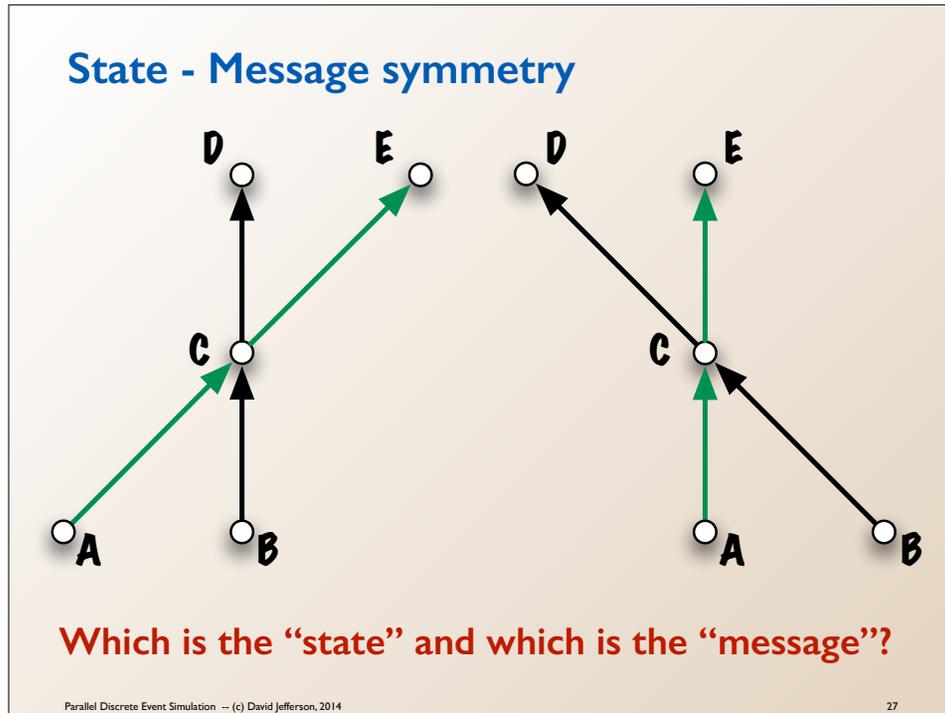
This diagram represents the before and after of the arrival of a straggler message.

Object A is at simTime 115 when an event message arrives in the past with a timestamp of 80. Two events at time 98 and 100 should not have been executed. The event at time 115 has been partially executed--we are still in the middle of executing it--and it may have sent some but not all of the event messages it would have. In any case, whatever it did is likely to be wrong.

We have to roll back to the state at time 80, i.e. to the state saved after execution of the event at time 64, the last correctly-executed event before time 80. The rollback consists of the following actions. Strictly speaking the kind of rollback we are describing here is called "aggressive cancellation", and is in some ways the most "optimistic" of the optimistic algorithms. An alternative is called "lazy cancellation", and another is called ??

The rollback consists of the following steps.

- 1) Insert the straggler message into the input queue where it belongs in the sort. It may be an antimessage and annihilate with another message already in the input queue. That make no difference in the algorithm at all--antimessages are treated identically.
- 2) Interrupt the event in progress (115). Restore the state saved after event 64 as the current state of the object. Delete the two subsequently saved states created by events 98 and 100, since they are (probably) incorrect (and one of them has been partially modified by event 115). Note that in a rollback variation called "lazy re-evaluation" these states would not actually be deleted at this time--and hence it really is possible to have "future" states in the queue. And in another variation called "sparse state saving" where we don't save states between every two event, but do it less often than that, then we might have to roll back farther than time 80 restore to an even earlier state.)
- 3) From the output queue, find the antimessage to the messages sent incorrectly after time 80, dequeue them, and deliver them to their receivers--the same objects that the original (incorrect) positive messages went to. Note that this includes any messages sent by the event that was in progress (and was interrupted), in this case event 115. Surprisingly (!) that is all that is required to exactly undo the effects of those positive messages, whether they have been delivered yet or not, or have been processed or not, or have caused generation of a tree of further, probably incorrect, distributed computation. The fact that this antimessage mechanism works in all cases, and allows the simulation globally to make progress asynchronously, independent of the speed of execution of the objects or the latency of message delivery, and regardless of the possibility of many interacting rollbacks in progress simultaneously, is a key observation at the foundation of most optimistic methods. (However, some less aggressive variations, e.g. *risk free* algorithms (in Paul Reynolds' taxonomy) do not transmit event messages until they can be committed, and thus have no need for antimessage. This comment is a forward reference, and I don't know whether I will get back to it in the course.)
- 4) Set the simulation clock of the object back to time 80 and start executing events forward again.



While a state and a message are very different when viewed from the point of view of the source code of a model, from the point of view of abstract semantics they are almost indistinguishable -- it is a matter of frame of reference. Whether something is a state or a message is a matter of local spacetime coordinate system rotation.

This is why I endeavor to make the representation of states and of messages and their queues as identical as possible.

Asynchronous, distributed rollback? Are you serious???

- **Must be able to restore any previous state (between events)**
- **Must be able to cancel the effects of all “incorrect” event messages that should not have been sent**
 - even though they may be in flight
 - or may have been processed and caused other incorrect messages to be sent
 - to any depth
 - including cycles back to the originator of the first incorrect message!
- **Must do it all *asynchronously*, with *many concurrently interacting rollbacks in progress*, and *without barriers***
- **Must deal with the consequences of executing events starting in “incorrect” states**
 - runtime errors
 - infinite loops
- **Must guarantee global progress (if sequential model progresses)**
- **Must deal with truly irreversible operations**
 - I/O, or freeing storage, or launching a missile
- **Must be able to operate in finite storage**
- **Must achieve good parallelism, and scalability**

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

28

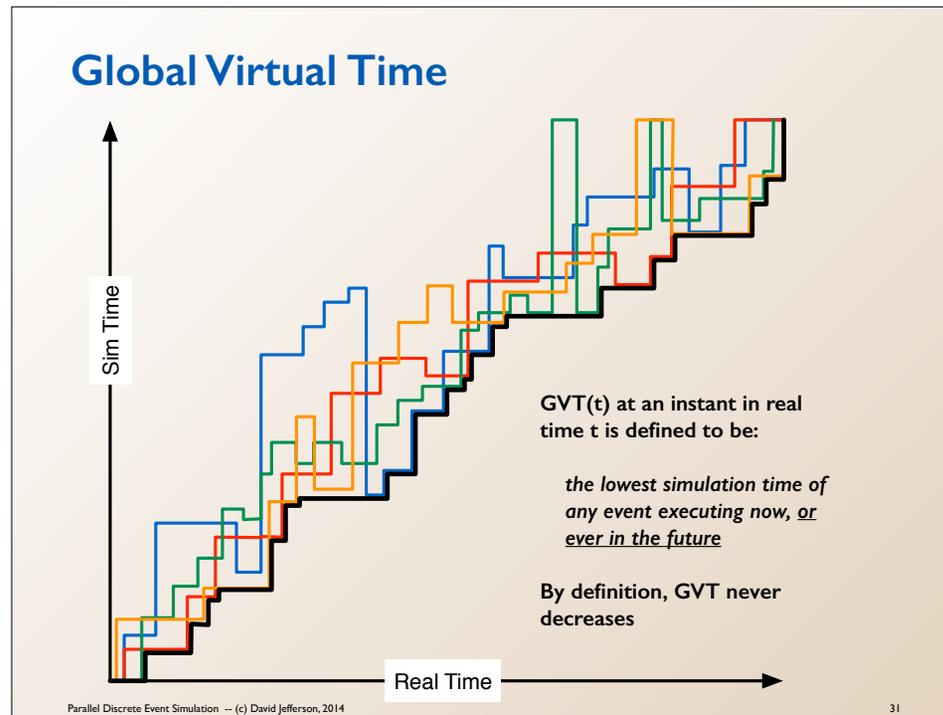
This shows the list of challenges we have to overcome for the Time Warp algorithm, or any optimistic PDES algorithm, to be practical. Most people with a background in asynchronous distributed computation who have not seen optimistic PDES algorithms are inclined to believe that doing this is either literally impossible, or at least hopelessly complex and slow.

Global Virtual Time (GVT) and Commitment

Local Virtual Time (LVT) and Global Virtual Time (GVT)

- **Virtual time, in the context of simulations, is usually the same thing a simulation time.**
 - Sometimes we define simulation time to be just the high order bits of virtual time.
- **The Local Virtual Time (LVT) of an object at a moment in the simulation measures how far the object has progressed simulation time, i.e. what its simulation clock reads.**
 - If an object is block because it has (temporarily) executed all of the events in its input queue, then we define its $LVT = \infty$
- **Global Virtual Time (GVT) measures how far the entire simulation has progressed globally and is (roughly) the minimum of all of the LVTs.**

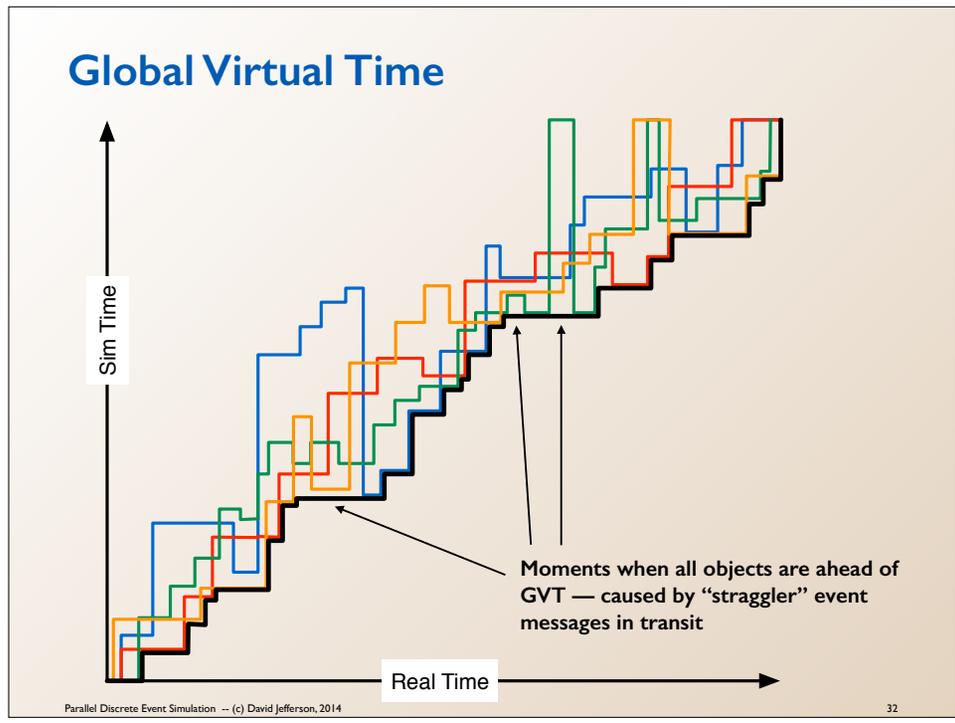
LVTs go forward and backward in time. GVT never goes backward, and is always a lower bound for all LVTs.



The *local virtual time* (LVT) of an object at an instant is the simulation time to which an object has progressed, i.e. what its simulation clock reads.

The colored lines each represent the LVT of one object in the simulation that goes forward and rolls back as the simulation executes.

The black line, acting as a tight monotonic lower bound envelope for all of the LVT curves, is Global Virtual Time, GVT.



There are a few places in this diagram where GVT is strictly lower than the minimum of all LVTs. That will happen at times when a message is in transit that happens to carry a lower timestamp (receive time) than the LVT of any object. When it is delivered, it will cause the receiving object to rollback to the message's received time.

Properties of Global Virtual Time

- **Events at virtual times lower than GVT can never be rolled back.**
- **GVT never decreases.**
 - In a well-posed simulation GVT inevitably *increases*.
- **GVT == ∞ is criterion for “normal” termination**

Parallel Discrete Event Simulation – (c) David Jefferson, 2014

33

By definition, GVT never decreases, and in fact it must increase unless the simulation has a bug in it like an infinite loop that would also affect the sequential execution of the same model.

GVT == ∞ if and only if all objects are at time infinity and no messages are in transit. In that case all objects have run out of events to process, and the entire global simulation terminates normally. Termination detection is the same as detecting the GVT == ∞ .

Definition of *instantaneous* Global Virtual Time

$$GVT = \min (LVT(p), RT(q), ST(r))$$

objects: p
forward messages in transit from p: q
reverse messages in transit from p: r

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

34

LVT(p) = Local Virtual Time, i.e. the simulation clock value of Object p

RT(q) = Receive Time of the (positive or negative) event message q that is in transit

ST(r) = Send Time of (positive or negative) event message r that is in transit in the reverse direction from receiver to sender (for storage management/flow control)

This is an *instantaneous* definition. It could be only calculated exactly if we stopped the simulation globally and waited for delivery of all messages. However, in practice, we calculate an estimate of it asynchronously, while objects are executing and messages are in transit.

At least half a dozen algorithms for estimating GVT and broadcasting the result without any barrier synchronization have been published. All take time $O(\log n)$ where n is the number of processes. They take advantage of the fact that a message may be in transit if it has been sent but not acknowledged yet (in a low-level reliable transmission protocol). Just because a message has not yet been acknowledged, that does not mean it has not actually been delivered yet--it may have been, but the ack has not yet arrived. In that case the message will be included in the "min" operation of both sender and received. But that does not change the estimated GVT value.

The estimate is guaranteed to be low, which is the direction you want it to be. An estimate that is too high would cause Time Warp to commit events that are not yet safe from rollback--that would be a disaster. But an estimate that is too low just delays the commitment of some events that are in fact safe to commit.